AFRL-RI-RS-TR-2009-265
Final Technical Report
November 2009

NEXUS OPERATING SYSTEM FOR TRUSTWORTHY COMPUTING

Cornell University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-265 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
PHILIP TRICCA
Work Unit Manager

/s/
MICHAEL J. WESSING, Deputy Chief
Information & Intelligence Exploitation Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
NOVEMBER 2009	Final	June 2007 – May 2009

4. TITLE AND SUBTITLE

NEXUS OPERATING SYSTEM FOR TRUSTWORTHY COMPUTING

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-07-2-0037

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Fred Schneider

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
04

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University, Office of Sponsored Programs
373 Pine Tree Road
Ithaca, NY 14850-2820

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIEB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-265

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-4550 Date Cleared: 26-October-2009

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The NEXUS project investigated new abstractions for building trustworthy applications. A new operating system was built, as well as a secure version of BGP and a suite of document management applications. A authorization logic, called NAL, was also developed; it provides the means to specify authorization policies in operating systems and in applications.

15. SUBJECT TERMS
Trusted computing, secure operating systems, credentials-based authorization, secure network protocols, external security monitors, network management systems.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Philip B. Tricca
U	U	U	UU	24	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of **Contents**

1. Introduction/Project Objectives

Trusted computing hardware, such as the Trusted Computing Group's *trusted platform module* (TPM), is increasingly being found in commodity processors. Such secure co-processors have the potential to create a revolution in software assurance by enabling (i) enforcement of strong guarantees about behavior and (ii) attribution of actions performed in those behaviors. Today's operating systems, however, neither exploit that new hardware nor provide alternatives for building such *trusted software*.

The Nexus project was intended to demonstrate how this new hardware could be used to build trustworthy applications. Specifically, we sought to

- define new abstractions suitable for supporting trusted computing,
- instantiate these abstractions in a prototype operating system, and
- validate the approach ideas by building trustworthy applications.

This agenda involved two threads of investigation. The first was concerned with building an operating system—Nexus—that itself is trustworthy. Such a task spans a broad set of issues, ranging from code security to the architecture and design of the kernel and *operating system* (OS) services. Our focus was more at the architecture and design end of the spectrum—we did program a prototype, but we had neither the time nor the resources to invest in building secure code (nor did we have new insights about such issues).

The second thread concerned using our Nexus operating system to support trustworthy applications. Here, the challenge was to select applications that were somehow representative and therefore increased our assurance in the generality of our schemes. We elected to explore networking and document-management applications. The networking applications enabled us to study the tension between performance and richness of security functionality, subject to the constraints of existing interfaces and structures. The document-management applications allowed us freedom to explore new, surprising functionality in a setting that was easy to motivate and explain.

2. Methods, Assumptions, and Procedures

The overall objective of the project was to produce insights—not code—and document those insights in technical papers, so that others could apply our research results in their settings. However, in the course of this project, we did implement several software prototypes. This ensured that problems were not being overlooked and it allowed us to demonstrate the feasibility of our approach. Needless to say, various implementation choices for our prototypes were dictated by expedience. For example, we used the C programming language (which is a poor choice for building secure code), and we did not write code anew when there was extant code we could modify more quickly.

Methods, assumptions, and procedures are further discussed under our results below.

3. Results and Discussion

3.1. Nexus Kernel Architecture

We implemented a prototype Nexus kernel. It supports a process-like abstraction called an *independent protection domain* (IPD), several forms of encrypted storage, and has a novel *input /output* (I/O) device driver architecture. That device driver architecture is now being studied by Microsoft, since it solves a problem they are wrestling with.

Device drivers typically are positioned inside an OS kernel and execute in supervisor mode, so they must be fully trusted. For Nexus, we developed a system architecture in which device drivers are located outside of the kernel and run without supervisor privileges. We employ hardware isolation and a form of reference monitor (in the kernel) to check the behavior of each driver against a corresponding safety specification. We summarize the approach in what follows; see [1] for details.

Even in user space, device drivers execute hardware I/O operations and handle interrupts. These operations can cause device behavior that compromises the integrity or availability of the kernel or other programs. Therefore, our driver architecture introduces a global, trusted *reference validation mechanism* (RVM) that mediates all interaction between device drivers and devices. The RVM invokes a device-specific *reference monitor* to validate interactions between a driver and its associated device, thereby ensuring the driver conforms to a *device safety specification* (DSS), which defines allowed and, by extension, prohibited behaviors. So each device driver is given access only to the minimum resources and operations necessary to support the devices it controls (least privilege), thereby shrinking the *trusted computing base* (TCB).[1] A system in which device drivers have minimal privileges is easier to audit and less susceptible to Trojans in third-party device drivers.

[1] Some drivers, such as the clock, provide functionality needed for defining or enforcing security policies. These device drivers remain part of the TCB no matter where they execute.

The DSS is expressed in a domain-specific language that we developed and defines a state machine that accepts permissible transitions by a monitored device driver. We built a compiler to translate a DSS into a reference monitor implemented by a state machine. Every operation by the device driver is vetted by the reference monitor, so operations that would cause an illegal transition are blocked.

The RVM protects the integrity, confidentiality, and availability of the system, by preventing:

- *Illegal reads and writes*: Drivers cannot read or modify memory they do not own.
- *Priority escalation*: Drivers cannot escalate their scheduling priority.
- *Processor starvation*: Drivers cannot hold the *central processing unit* (CPU) for more than a pre-specified number of time slices.
- *Device-specific attacks*: Drivers cannot exhaust device resources or cause physical damage to devices.

In addition, given a suitable DSS, the RVM can enforce site-specific policies to govern how devices are used. For example, administrators at confidentiality-sensitive organizations might wish to disallow the use of attached microphones or cameras; or administrators of trusted networks might wish to disallow promiscuous (sniffing) mode on network cards.

In our user-space driver architecture, drivers, like any other user process, are loaded from a file system; once loaded, they execute and can be unloaded and restarted at any time. When a driver is first loaded, it executes a system call to find a compatible device. As part of this system call execution, the RVM identifies an appropriate device and reference monitor and returns to the driver a structure describing the device ID and I/O-resource assignments. The driver then uses driver system calls to perform I/O operations and receive interrupts. Subsequent uses of those calls cause the RVM to invoke the reference monitor. Reference monitors are instantiated immediately after endpoint enumeration, based on device IDs. Reference monitors persist, even if corresponding drivers are unloaded and restarted

Drivers are not trusted, but the RVM, reference monitors, and devices are. Moreover, reference monitors are compiled from DSSes, so DSSes and the DSS compiler must be trusted. Some DSSes will be written by hardware manufacturers; others will be written by independent experts, including security firms or OS distributors. But independent of the source, a DSS ought to be small and declarative. Further, because they describe devices, not drivers, there need only be one DSS per device. Hence, they are conducive to auditing.

We assume devices behave safely if given sufficiently restricted inputs. Such an assumption is inescapable, because devices can access any memory, generate arbitrary interrupts, and starve hardware buses directly. The two sources of driver misbehavior we considered are drivers designed by malicious authors (Trojans), and drivers with bugs that can be subverted by users or remote attackers. Both are dealt with by our RVM.

The RVM prevents drivers from performing invalid reads and writes using hardware isolation and by checking driver accesses to *direct memory access* (DMA) control registers.

- Hardware isolation works as with other user processes, giving each driver process direct access only to its own memory space.

- By checking that every DMA address sent to the device is allocated to the driver, the RVM prevents a device driver from using DMA for illegal reads and writes.

The RVM must also defend against a device driver that attempts to escalate its execution priority or that starves other processes and the kernel by causing large numbers of interrupts or by spending too much time in high-priority interrupt handlers. A timer driver might set too high a timer frequency, or a sound card driver might set too small a DMA buffer for playback, causing frequent notifications to be generated when the buffer becomes empty.

Some of these unacceptable behaviors can be prevented when the driver is setting up the device; for example, by a reference monitor imposing a lower bound on the sound card DMA buffer size. But RVMs provide three additional protection measures. First, the RVM limits the frequency at which a driver can receive interrupts, with different limits for different types of devices. Second, the RVM limits the length of time that an interrupt handler runs. Third, the RVM ensures that each interrupt handler acknowledges every interrupt, to prevent devices from issuing additional interrupts for the same event. Finally, an RVM must prevent invocations of operations known or suspected to harm devices. Examples include: overclocking processors, sending a monitor an out-of-range refresh rate, instructing a disk to seek to an invalid location, or writing invalid data to non-volatile configuration registers. Other attacks against devices involve exhausting finite resources, such as wearing out flash memory with excessive writes or wasting battery power on mobile devices. The RVM prevents many such attacks by allowing only well-defined operations at rates presumed to be safe.

While the RVM approach is general enough to enforce rich safety properties, we do not anticipate that RVMs will be used to enforce driver semantics expected by applications. Our reference monitor implementations do not, for example, ensure that network drivers only send legal *transmission control protocol* (TCP) packets. They also do not prevent a malicious driver from providing incorrect or incomplete access to a device (i.e. denial of service). Such protections concern end-to-end properties, hence we believe that they are best implemented above the driver level.

We implemented user-space device drivers for the i810 sound card, e1000 network card, *universal serial bus* (USB) *universal host controller interface* UHCI controllers, USB mice, and USB disks in Nexus. We quantified the ease of driver porting and the auditability of DSSes by counting the number of lines of code in each DSS and the number of lines changed to port each Linux driver to Nexus. The number of changed and added lines was small.

We wrote each DSS by referring to the manufacturer's documentation about device behavior and to existing drivers. The DSS for USB UHCI was derived entirely from the documentation. The i810 and e1000 DSSes were based on documentation that describes features our drivers actually use; other features are disallowed by the DSS. Writing a DSS based on an existing driver is tempting, but risks disqualifying other drivers that attempt different (but safe) behavior. Writing a DSS based on all features described in published documentation is more time-consuming, but in theory, it admits any legal driver. Based on our experience, we estimate the time to develop a DSS, given a working driver, manufacturer's documentation, and familiarity with the DSS language but not with the device, as one to five days

Our Nexus drivers exhibited performance comparable to in-kernel, trusted drivers, with a level of CPU overhead acceptable for most applications. For example, the monitored driver for an Intel e1000 Ethernet card has throughput comparable to a trusted driver for the same hardware under Linux. And a monitored driver for the Intel i810 sound card is able to provide continuous playback. Drivers for a disk and a USB mouse have also been moved successfully to operate in user space with safety specifications.

Accepted quantitative metrics for the security of a system do not exist. Nevertheless, to establish the security of our RVM and reference monitors, we used two approaches others have used. First, we simulated unanticipated malicious drivers by randomly perturbing the interactions between drivers and the RVM, resulting in potentially invalid operations being submitted to the reference monitor and possibly to the device. Second, we built specific drivers that perpetrate known attacks on the kernel using interrupt and DMA capabilities.

We simulated unanticipated malicious drivers by changing operations and operands in a layer interposed between a legal driver and the RVM. This layer modified each operation according to an independent probability of 1 in 16,384. Each operation was a read or a write; our modifications involved replacing either the address, the length, or the value (at random) with another value in the appropriate range. So, a write to an I/O port was replaced with a write to a port in the same range, a write of a different length, or a write of another value. Reads were perturbed similarly. Note, this approach does not produce repeatable experiments, because driver behavior depends on external factors like the OS scheduler and the arrival times of packets, which are not under our control.

We applied perturbation testing to the e1000 driver. When the modifications were benign, the driver showed no apparent failures. Sometimes, the driver itself detected an error (e.g., a status register read failed a sanity check) and exited cleanly. Often, the reference monitor detected an illegal operation, and the RVM terminated the driver. Finally, our perturbations sometimes caused the driver to get out of sync with the device, after which no further packets were sent or received. This does not compromise the integrity or availability of the kernel or the device, so the RVM has no obligation here. An unmonitored driver completed more tests with no apparent failure than a Safe driver (i.e., a driver being monitored with a suitable DSS) did, because the reference monitor used for the Safe driver blocks all unknown behaviors—even if it might be benign.

We hoped the perturbed unmonitored driver would cause kernel livelock, starvation, or a crash. In practice, however, the likelihood of causing driver crashes and stalls is much higher. The 31st run of this test rendered the device unusable: neither the Linux nor the Nexus driver could thereafter initialize the card.

In addition to perturbation testing, we wrote several malicious drivers to execute specific attacks on the kernel using the e1000's interrupt and DMA capabilities:

- *Livelock*: The driver never acknowledges interrupts, resulting in a flood of interrupt activity and starvation for all other processes.

- *DMA kernel crash*: The driver uses the device to write to kernel memory, resulting in a system crash

- *DMA kernel read*: The driver sends a sensitive page (e.g., containing a secret key) to a remote host.

- *Direct kernel read/write*: The driver constructs a pointer and reads or writes sensitive data directly.

- *DMA kernel code injection*: The driver points a DMA buffer pointer at system call code, then pings a remote machine with attack code. The response is written over the target system call implementation. The attacking driver then invokes the system call to gain control of the kernel.

- *DMA read/write to other device*: The driver uses a ping to overwrite video memory, resulting in an image appearing on the screen.

Not surprisingly, the livelock and DMA attacks succeed when run as unmonitored drivers, all the attacks succeed on drivers in the kernel, and they are all are caught by the RVM when it monitors a driver. The livelock attack is prevented by the RVM terminating any driver that does not acknowledge the interrupt by reading the interrupt control register. The DMA attacks are prevented by the RVM terminating any driver that attempts to transmit or receive packets with any invalid addresses in the transmit or receive buffer lists. Finally, any direct attempt to read or write the memory of other drivers is blocked by hardware isolation in all modes except Kernel.

3.2 Trustworthy Network Protocols

The techniques we employed in building trustworthy device drivers actually turn out to be quite general. The key ingredients are (i) a reference monitor inserted into the output channel of a target component, (ii) a clear specification of secure behaviors for that component (perhaps in terms of inputs or the state of its environment), and (iii) a strong isolation mechanism to separate the reference monitor from the target. We illustrated this generality in [2] by using the same basic approach to implement a trustworthy version of the Internet's *border gateway protocol* (BGP).

BGP is the protocol routers use to announce, propagate, and withdraw routes between *autonomous systems* (ASes) in the Internet. An AS is a portion of the network presumed to be under a common administrative control. A *BGP speaker* is any router that participates in BGP; usually, it is a router at the edge of an AS. BGP speakers in an AS maintain *transmission control protocol* (TCP) connections to *peers*—BGP speakers at other ASes with which the AS has peering relationships. A BGP speaker is connected to its peers directly or by statically
configured routes.

Each AS controls a subset of all *internet protocol* (IP) addresses, represented as one or more IP address prefixes—contiguous sets of IP addresses with a common string of leading bits. Each BGP speaker maintains a table mapping IP prefixes to next-hop routing information. BGP speakers disseminate and discover this routing information by announcing their own IP prefixes and by receiving similar announcements from BGP speakers in peer ASes. A BGP speaker's configuration lists all prefixes it originates, other BGP speakers that are its peers, and policies for choosing a preferred route to each prefix.

Our secure version of BGP is called N-BGP. We built it to illustrate the utility of employing an *external security monitor* (ESM), which is a new kind of network component we developed for securing legacy protocols without requiring modifications to existing hardware, software, or the protocol. An ESM is a separate host that checks each message sent by a legacy host against a safety specification. There are two ways to add an ESM to a network.

- A *proxy ESM* explicitly filters and forwards relevant traffic between a target and all other hosts. An administrator must change the target's configuration—but normally not its software implementation—to send relevant traffic to the ESM rather than directly to a peer. Unrelated traffic (in the case of BGP, all data traffic) is not sent through the ESM.

• A *sniffer ESM* passively captures packets on all network links connected to some target. Because a sniffer ESM must capture all traffic into and out of its target, it is inefficient in cases where most of that traffic is not relevant to the monitored protocol. A sniffer ESM cannot slow or break the underlying protocol, and the underlying protocol continues (albeit unmonitored) if the ESM fails. But a sniffer ESM cannot block invalid traffic. Sending alerts on the security plane is the only way for a proxy ESM to make use of judgments about message validity.

Proxy and sniffer ESMs each have advantages and disadvantages. A sniffer ESM might not have sufficient capacity to isolate monitored protocol traffic on a backbone link; a proxy ESM receives only this traffic and thus is suitable for links of any speed. Sniffer and proxy ESMs interoperate, forming a single security plane.

ESMs use trusted hardware to assure remote principals that the safety specification is being enforced, use physical isolation to ensure integrity for the reference monitor, and use an overlay network to alert each other about invalid behavior and to initiate remedial actions. When run on commodity hardware, we found N-BGP was fast enough to monitor a production BGP router. And by running simulation experiments we established that deploying N-BGP at a random 10% of autonomous systems in the Internet suffices to guarantee security for 80% of Internet routes where both endpoints are monitored by N-BGP.

Because ESMs do not have to implement complete protocol functionality or provide a rich user interface, they can have a substantially smaller trusted computing base than the targets they monitor. And because ESMs enforce a safety specification on network messages, a single ESM implementation is compatible with any implementation of a protocol, regardless of software version, configuration, or policy. Finally, because ESMs deployed in different administrative domains communicate using an overlay, ESMs have access to information not available at any one target, and ESMs can globally coordinate remedial actions.

Our N-BGP defends against false-origination and path-truncation attacks, which give rise to BGP route hijacking, traffic stealing, and black holes. Although these attacks have been well known for more than ten years, today they are increasingly being exploited by spammers. Each N-BGP host intercepts all BGP messages received and sent by a single target BGP router and checks them against a safety specification that characterizes route advertisements the target may send given the route advertisements it has received. For example, a router that has received routes no shorter than n hops for a given remote destination should not announce routes shorter than n+1 hops for that destination. A shorter advertisement indicates a path truncation (also known as a black hole or traffic stealing) attack, and it will trigger NBGP to take remedial action locally, by notifying the site administrator, and remotely, by purging the offending route advertisement from the network.

3.3 Network Configuration Management

Although a network's configuration can have a significant impact on the performance, robustness, and security of applications, today's networks lack support for reporting network configuration differences. Our NetQuery system [3] aims to correct this by implementing a trustworthy channel for disseminating the properties of networks and their participants. Specifically, NetQuery implements a distributed, decentralized, tuple-based attribute store that records information about network entities. Operators can add new tuples into this store and can also annotate existing tuples with new, custom attributes, thus allowing the system to support network entities and properties not anticipated at the time of deployment. NetQuery clients can query the attribute store for the current network state and can install event triggers to detect future state transitions, thus establishing long-running guarantees over the behavior of the network.

NetQuery is concerned with *network entities*, which include physical devices such as routers, switches, and end hosts as well as logical entities, such as flows and applications. Each network entity has an associated set of properties, which are represented in NetQuery as attribute/value pairs. These properties may be intrinsic to a device (such as a router's routing tables) or they may be arbitrary labels assigned by third parties (such as a certificate from an audit service asserting that a router is properly configured).

While NetQuery provides a globally unified interface, the implementation and storage of attribute/value pairs is decentralized. Every network operator deploys a NetQuery server dedicated to storing attribute/value pairs for its portion of the network and specifies an access policy to these attribute/value pairs. NetQuery enables anyone to tag any entity with an arbitrary property. The tuplespace NetQuery implements may thus contain conflicting information but this is addressed by allowing applications to ignore information from sources they don't trust. Specifically, a policy language enables applications to assert its trust in an attribute/value pair and to control access to proprietary information and to sensitive operations.

We assume all networked components are equipped with secure hardware co-processors to serve as a root of trust for claims made by the network component. Admittedly, this is a "clean slate" design, since today's network components do not have such secure co-processors. But we predict that limitation will be short-lived, because *trusted platform modules* (TPMs) provide a cheap way to support the creation of the unforgeable certificates that are the only sensible foundation on which to base a trustworthy network. However, we have also investigated how to integrate current systems equipped with management interfaces, such a *simple network management protocol* (SNMP) but no secure co-processor.

Three scenarios we investigated illustrate the value that NetQuery can provide.

• *Checking end hosts.* Misconfigured end hosts can compromise the integrity of a network. A NetQuery-enabled network can restrict access based on the configuration of end hosts. For instance, before allowing a newly connected end host to send packets, a switch could verify that the end host is running the latest software versions and a virus checker.

• *Checking paths.* Middleboxes that can perform deep packet inspection for large volumes of network traffic enable *internet service providers* (ISPs) to monitor and modify streams that traverse their network. A privacy-conscious user who wants to know how data from her web sessions will be monitored can use NetQuery to discover whether there are entities that can potentially access and record her sessions and, if applicable, to obtain guarantees from the network on how packets are handled.

• *Differentiating providers.* Customers currently use reputation as the primary way they differentiate ISPs. And NetQuery enables ISPs to advertise the performance and robustness features that they provide. For instance, a wireless service provider can use NetQuery to advertise its backhaul capacity and traffic management techniques; clients can use this to select the best available network.

3.4 Beyond Credentials-Based Authorization

In *credentials based authorization*, requests to access a resource or obtain service are accompanied by *credentials*. Each request is either authorized or denied by a *guard*, which uses the accompanying credentials (perhaps augmented with other credentials or information about the state) to make that decision and enforce some given security policy. Authorization decisions are thus decentralized, with accountability of each element in the decision made explicit and with authority shared among the guard and the principals who issue credentials. This decentralization means the regime is ideally suited for use in distributed systems.

An untrustworthy principal might attempt accesses that violate a security policy, whereas (by definition) a trustworthy one wouldn't. So a guard ideally should authorize only those requests made by trustworthy principals. However, determining the trustworthiness of a principal is rarely feasible, and guards typically substitute something that is easier to check.

• *Axiomatic basis.* With guards that use an access control list, we accept on faith that all principals appearing on the access control list are trustworthy, so the guard authorizes requests made by these principals. Axioms are statements that we accept on faith, so we might label this an *axiomatic basis for trustworthiness*. The same applies when a system uses some form of reputation to decide whether a principal's request should be authorized. An axiomatic basis also is implied when a guard authorizes loading and running an executable only if the value of a hash indicates that the executable is unaltered from what comes in some standard distribution or if a digital signature establishes that the executable was generated by some approved software provider.

• *Analytic basis.* Analysis provides a way to predict whether certain behaviors by a program *P* are possible, and some guards employ an *analytic basis* for authorizing requests from principals executing *P*. Specifically, an analysis establishes that *P* can be trusted not to commit certain abuses and, therefore, granting the request cannot enable *P* to violate the security policy. Proof carrying code is perhaps the limit case. Here, a program *P* is accompanied by a proof that its executions satisfy certain properties; a request to execute *P* is authorized if and only if a proof checker trusted by the guard establishes that the proof is correct and that the property proved is sufficiently restrictive. As another example, some operating systems will authorize a request to load and execute code only if that code was type checked; type checking is a form of analysis, and programs that type check can be trusted not to exhibit certain malicious behaviors.

• *Constructive basis.* A *constructive basis* for authorization is involved whenever a program is transformed prior to execution so that it can be trusted in ways the original could not. Examples of this approach include sandboxing, *software fault isolation* (SFI), in-lined reference monitors, and other program rewriting methods.

The discussion above suggests that no single basis for establishing trust is used for all guards and security policies. Moreover, it seems reasonable to believe that no single basis for trust would suffice within even a single guard when authorization decisions are being made in a decentralized way. This led us to conjecture that substantial benefits could come from developing an authorization framework that incorporates and unifies the axiomatic, analytic, and constructive bases for trust. So we designed such a framework and implemented it for Nexus. Specifically, we developed a language and logic NAL (Nexus Authorization Logic) for specifying and reasoning about credentials and security policies [4]. NAL builds on Abadi's CDD access control logic by adding support for axiomatic, analytic, and constructive bases for trust and by adding new kinds of principals (groups and sub-principals) that help bridge the gap from the simplifications and abstractions found in CDD to the pragmatics of actual software applications.

Besides implementing support for credentials and guards in Nexus, we built a suite of document-viewer applications. With each, the viewer application implements a guard and the document (not the human user, as might be expected) is the principal that issues requests for document display. CertiPics (Certified Pictures) enforces the integrity of displayed digital images by imposing chain-of-custody restrictions on the production pipeline; TruDocs (Trustworthy Documents) controls the display of documents that contain excerpts whose use is subject to restrictions; and ConfDocs (Conf Documents) protects confidentiality of documents built from text elements having security labels.

Principals in NAL. Distinct *Nexus Authorization Logic* (NAL) principals are assumed to have distinct names, with the name of a principal inextricably linked to the worldview of that principal. Naming schemes that satisfy these assumptions include:

- Use a public key as the name of a principal, where that principal is the only entity that can digitally sign content using the corresponding private key. A principal named by a public key adds a belief *S* to its worldview by digitally signing an encoding of *S*. So, a digitally signed representation of the NAL statement *S*, where public key K_A verifies the signature, conveys NAL formula: K_A **says** *S*.

- Use the hash of a digital representation of a principal as the name of that principal. Thus, a principal *Obj* is named by hash *H*(*Obj*), and the principal includes a belief *S* in its worldview by having an encoding of *S* stored at some well known position in *Obj* . So *Obj* conveys the NAL formula *H*(*Obj*) **says** *S* by having *S* be part of *Obj*.

Public keys are attractive for naming principals because a credential conveying K_A **says** *S* can be forwarded from one principal to another without any need to trust intermediaries—after forwarding, the digital signature continues to identify the principal that originally created the credential. But public-private key pairs are expensive to create. Moreover, private keys can be kept secret only by certain types of principals. With a TPM, you can associate a private key with a processor and keep it secret from all software that runs on the processor; without a TPM, you can associate a private key with a processor but keep it secret only from non-privileged software. And there is no way to associate a private key with a non-privileged program executing on a processor yet have that key be secret from the processor or from privileged software being run.

Hashes are relatively inexpensive to calculate and do not require keeping secrets. But an object *Obj* and credential conveying *H*(*Obj*) **says** *S* can be easily and undetectably changed into an object *Obj*' that instead conveys *H*(*Obj*') **says** *S*': locate and replace *S* in *Obj* by *S*', and then compute the new name *H*(*Obj*'). Also, note that a principal named by a public key can revise its worldview and create corresponding credentials at any time, whereas a principal named by a hash cannot.

NAL is agnostic about what schemes are used to name principals. Our experience with building applications for Nexus suggests that public keys and hashes both have uses. Nexus also implements various specialized naming schemes for some of its abstractions (e.g., processes) that serve as principals.

The decision to authorize a request can be expressed as a question about formula derivation in NAL. An access request by a principal *A* is modeled using (i) NAL formula *A* **says** *S* to convey request particulars, (ii) NAL formulas *C1*, *C2*, ..., *Cn* for accompanying credentials and (iii) NAL formula P_G for the authorization policy being enforced by guard G. The request is granted if and only if G determines that P_G can be derived from

(*A* **says** *S*) **and** *C1* **and** *C2* **and** ... **and** *Cn*

using NAL's inference rules; otherwise the request is denied.

With this setup, not only does the guard make an authorization decision but, through the derivation for P_G, the guard documents a rationale for granting the request and makes clear the role each credential has played. The derivation is thus a form of audit log—and a particularly descriptive one, at that. The wide range of possible implementations for this derivation-based approach to authorization gives system designers considerable flexibility to make engineering trade-offs when implementing guards. Decisions the designer must make include:

- Where is each credential stored? Credentials could be stored at the requesting principal, at the guard, or elsewhere in the system.

- How is each credential obtained by the guard? Credentials could accompany a request, be fetched when needed by the guard, or be sent periodically to the guard.

- Where and how is the derivation of the guard's authorization policy P_G performed? This could be done by the requesting principal, it could be done by the guard (perhaps by coordinating a distributed computation based on subgoals in the proof), or it could be a service provided by some trusted third party.

- How and when is each credential generated? If a credential corresponding to NAL formula *Ci* is issued, then we might expect *Ci* to hold thereafter. But changes to the system state could cause a principal to change its beliefs, falsifying *Ci*. Guards and other principals with access to the credential but lacking independent means for validating formula *Ci* must be implemented with this reality in mind.

Sources of Derivations. Constructing a NAL derivation for some arbitrary given formula is an undecidable problem, because NAL terms include integers and rich data structures whose axiomatizations are undecidable. However, NAL formulas P_G for authorization policies found in practice are often easily derived when accompanying credentials have a prescribed form. For example, we might specify discretionary access control for requests from a principal *A* to access an object *obj* by writing the following NAL formula for P_G:

$$(A \textbf{ says } \text{access}(obj)) \quad \textbf{and} \quad (A \rightarrow \text{owner}(obj))$$

Derivation of this is trivial if we prescribe that requests *A* **says** access(*obj*) are accompanied by a credential that attests owner(*obj*) **says** *A* → owner(*obj*). Or, the guard for *obj* might itself store an access control list ACL_{obj} , which is interpreted as attesting owner(*obj*) **says** *A* → owner(*obj*) for every principal *A* appearing in ACL_{obj} .

An alternative to having a guard G perform a derivation of P_G would be to have G check a derivation supplied with the request. This is a decidable task because, by definition, inference rule applications are mechanically checkable. To illustrate, we return to the discretionary access control given above. Instead of accompanying a request with a credential that attests to the needed delegation, a principal *A* making a request might provide a set of credentials and a derivation from those credentials of what is needed for establishing conjunct *A* → owner(*obj*).

The idea of having derivations accompany requests is not a panacea. In order for a principal to produce a derivation of P_G , that principal must know what P_G is, which requires divulging the criteria for authorizing requests. And sometimes we may want to keep principals unaware that different criteria apply to each. Also, having each different requester independently perform the derivation of P_G makes changing P_G difficult, since all principals that submit requests to guard G would have to be identified and updated.

Credential and Policy Dynamics. Possession by guard G of a credential puts the NAL formula *C* conveyed among G's beliefs. Therefore, NAL models G's possession of a credential as: G **says** *C*. One might hope that

$$(\text{G } \textbf{says } C) \supset C$$

would hold as well, although this is by no means guaranteed. The principal issuing a credential might well subsequently change its beliefs (perhaps because the state has changed) but after a credential has been sent elsewhere, that credential is no longer available to the issuer for update or deletion. Yet if (G **says** *C*) $\supset$ *C* does not hold for one or more credentials that a guard G has received, then the guard could have a NAL derivation of authorization policy P_G from those credentials even though P_G does not actually hold.

Even if a guard G could check the truth of NAL formula *C* conveyed by each credential that G uses, there is no guarantee that P_G would hold after this checking. The process of checking the truth of *C* will take time, and concurrent actions elsewhere in the system could falsify the formula conveyed by one credential while the others are being checked. However, by restricting system execution, guard construction, and/or what NAL formulas credentials may convey, we can ensure that P_G will hold whenever it can be derived in NAL from a request and accompanying credentials.

System execution generally satisfies certain restrictions—time never decreases and the past is immutable—not to mention restrictions coupled to the semantics of various system functionality and applications. This means that some truths do not change as execution proceeds, and this can be leveraged for defining credentials that cannot be falsified by future execution. By imposing additional restrictions on execution by principals that falsify certain predicates, we obtain a second general approach for constructing credentials that satisfy (G **says** *C*) $\supset$ *C*.

Implications for Privacy. Credentials used for authorizing a request convey, hence reveal, attributes of the requester. This can impinge on privacy, where *privacy* is defined as the right of an individual to control the dissemination and use of information about that individual. To protect privacy, we should strive to employ authorization policies that minimize the information needed about individuals.

Credentials-based authorization offers the flexibility to define policies and credentials involving only limited information about individuals. So credentials-based authorization provides what is needed for protecting privacy through system designs that instantiate the following principle, where an identity is a set of attributes.

> *Principle of Minimal Identity*. Employ identities that embody the smallest set of attributes needed for the task at hand.

By way of comparison, in identifer-based authorization, access decisions depend only on a label associated with the principal making the request—there is no flexibility to disclose only certain attributes of the requester (although one can employ labels whose connection to an individual is impossible discern). Privacy with identifer-based authorization is, consequently, coarser-grained. Identifer-based authorization also admits privacy compromises based on correlating the labels used in different requests; credentials-based authorization does not, because correlated credentials do not imply correlated requesters.

The contents of a credential can be seen as a kind of privilege, since it provides a basis for authorizing requests. Logical implication defnes a partial ordering on these privileges: if $C \supset C'$ holds, then a credential *A* **says** *C* is considered stronger than *A* **says** *C'*. We might then view credentials-based authorization through the lens of the well known Saltzer-Schroeder mandate:

> *Principle of Least Privilege.* Assign each principal the minimum privileges it needs to accomplish its task.

And, in so doing, we discover this mandate offers the same guidance as the Principle of Minimal Identity! So with credentials-based authorization, security and privacy are both well served by expecting weaker credentials from principals. This account also clarifies that merits of preferring analytic and constructive bases for authorization to an axiomatic basis. The analytic and constructive bases allow credentials to embody exactly what is needed for authorization, and are thus consistent with the Principles of Minimal Identity and Least Privilege. The axiomatic basis, however, is ultimately a form of identifer-based authorization and, therefore, it is going to be coarse-grained, subject to linking attacks, and unlikely to satisfy either the Principle of Minimal Identity or the Principle of Least Privilege. There are thus some strong, principled arguments in favor of the authorization architectures that Nexus supports.

4 Conclusions

Under the auspices of this funding, progress was made toward realizing the goal of constructing an operating system to support the construction of trustworthy applications. Additional work is needed to complete the system and test the ideas. But much of the research involved ideas that transcend any single operating system implementation, and we expect to see these ideas used in other systems.

5 Publications

[1] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation OSDI '08* (San Diego, CA, December 2008), 241–254.

[2] Patrick Reynolds, Oliver Kennedy, Emin Gun Sirer, and Fred B. Schneider. Using external security monitors to secure BGP. Submitted for publication to *ACM/IEEE Transactions on Networking*.

[3] Alan Shieh, Oliver Kennedy, Emin Gun Sirer, and Fred B. Schneider. NetQuery: A General-Purpose Channel for Reasoning about Network Properties. Technical Report http://hdl.handle.net/1813/12714, eCommonsCornell. May 15, 2009.

[4] Fred B. Schneider, Kevin Walsh, and Emin Gun Sirer. Nexus Authorization Logic (NAL): Design Rationale and Applications. Submitted for publication. August 2009.

6. Presentations

Besides I-ARPA PI meetings, the following public presentations have been made regarding work supported by this contract.

1. Fred B. Schneider: Credentials-based authorization: Evaluation and implementation. Keynote lecture, ICALP 2007: 34th International Colloquium on Automata, Languages and Programming (Wroclaw, Poland, July 2007).

2. Fred B. Schneider: Credentials-based authorization: Evaluation and implementation. Microsoft Corporation (Redmond, Washington, July 2007)

3. E. Gun Sirer: Trustworthy Computing with the Nexus Operating System. University of Washington (Seattle, Washington, Dec. 2008).

4. Patrick Reynolds: Device Driver Safety Through a Reference Validation Mechanism. OSDI'08 (San Diego, California, Dec 2008).

5. Fred B. Schneider: Minimal Identity: Authorization and Accountability. Keynote lecture, Young Engineering Scientists Symposium (YESS) 2009. (Washington D.C., July 2009).

7. List of Acronyms and Terms

ACL: Access Control List
AS: Autonomous System
BGP: Border Gateway Protocol
CertiPics: Certified Pictures
ConfDocs: Conf Documents
CPU – Central Processing Unit
DMA: Direct Memory Access
DSS: Device Safety Specification
ESM: External Security Monitor
I/O: Input/Output
IP: Internet Protocol
IPD: Independent Protection Domain
ISP: Internet Service Provider
NAL: Nexus Authorization Logic
OS: Operating System
RVM: Reference Validation Mechanism
SFI: Software Fault Isolation
SNMP: Simple Network Management Protocol
TCB: Trusted Computing Base
TCP: Transmission Control Protocol
TPM: Trusted Platform Module
TruDocs: Trustworthy Documents
UHCI: Universal Host Controller Interface
USB: Universal Serial Bus